Rapport de Projet : Do you want some " Chunks " ?

François Chapuis, Roman Mkrtchian, Frédéric Payan, Marc Antonini

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR7271

Rapport de Projet
Do you want some ç Chunks è

Fran-ois Chapuis, Roman Mkrtchian, Fr®d®ric Payan, Marc Antonini

MEDIACODING

Rapport de Recherche
ISRN I3S/RR-2013-06-FR

Mai 2012

POLYTECH
NICE-SOPHIA
Département Sciences Informatiques

Université Nice Sophia Antipolis

Rapport de Projet : Do you want some « Chunks »

Projet de compression d'objets 2D

Par François Chapuis et Roman Mkrtchian

2011/2012

Table des matières

Introduction

Les industriels de l'électronique utilisent des fichiers représentant les formes présentes sur les circuits imprimés. Ceux-ci peuvent comporter plusieurs milliards de formes. Il s'agit dans ce projet de créer un logiciel permettant de compresser ces fichiers sous forme binaire, de pouvoir les décompresser et de pouvoir observer graphiquement la version compressée et non compressée en mettant graphiquement en évidence le type de compression utilisée.

Partie parsage du fichier texte

La disposition des composants électroniques sur un circuit imprimé est décrite dans un fichier texte. Chaque ligne de ce fichier décrit un composant. On trouve, au début de chaque ligne le type de composant décrit (Rectangle, Polygone...) puis une liste de valeurs décrivant les caractéristiques du composant, séparées par des espaces. Par exemple, dans le cas d'un rectangle, les valeurs spécifiées sont les coordonnées du point en bas à gauche et celles du point en haut à droite.

Exemple de ligne d'un fichier : RECT -0.630 0.040 -0.590 0.060

La première étape a été de lire ("parser") ce type de fichiers et d'effectuer des calculs simples afin de déterminer le type précis des formes. Cela permet de définir les informations permettant de décrire de la manière la plus concise possible la forme.

Par exemple, il faut **4 valeurs** au minimum pour représenter un rectangle quelconque (sa longueur, sa largeur, une coordonnée en abscisse, et une coordonnée en ordonnée). En revanche, si le rectangle est décrit comme "RECT -0.040 -0.030 -0.020 -0.010" on voit que :

Longueur = -0,020 - (-0,040) = 0,020

Largeur = -0,010 - (-0,030) = 0,020

Dans ce cas, on s'aperçoit que la forme lue est en réalité un carré. Dans ce cas, 3 valeurs suffisent : la longueur de l'arête (0,020 dans ce cas) et les coordonnées en abscisse et en ordonnée d'un point du carré.

Cette reconnaissance préliminaire permet déjà de réduire le nombre d'informations à stocker.

Partie visualisation

L'interface de visualisation a été réalisée en utilisant les bibliothèques Swing et awt. Le langage Java a été choisi en partie pour la simplicité à réaliser des interfaces graphiques avec celui-ci.

Il s'agissait d'afficher les stimuli tels qu'ils sont avant la compression, et de faire un autre affichage pour observer graphiquement les associations ayant été effectuées par la compression.

Nous avons d'abord permis la visualisation des composants du le circuit imprimé (stimuli) de manière brute, puis cette visualisation a été abandonnées au profit d'une autre légèrement plus évoluée colorant les stimuli en fonction de leur forme primitive (dans notre cas il n'y a que deux couleurs différentes, pour les carrés et les rectangles).

Nous avons ensuite coloré, dans une deuxième fenêtre graphique, les composants de même dimensions (donc ceux faisant référence au même chunk de dimension) de la même couleur. Pour cela nous avons du choisir de manière aléatoire un nombre de couleurs correspondant au nombre de chunks de dimension, répartis uniformément dans le spectre des couleurs visibles. Le code pour faire cela se trouve dans la classe utils.Graphic.java.

Il s'agissait de trouver un nombre défini de couleurs les plus espacées possible, en faisant varier les 3 couleurs de base Rouge, Vert et Bleu entre les valeurs 0 et 255.

Nous avons alors fait 3 boucles for imbriquées, faisant chacune varier une des composantes, et faisant chacune *racine cubique du nombre de chunks de dimension* itérations. Ensuite nous avons itéré sur ces valeurs et pris le nombre de valeurs nécessaires, en les espaçant le plus possible.

Ceci a permis de visualiser l'effet des chunks de dimension.

Pour finir, nous avons, sur ce deuxième affichage, entouré les stimuli liés par un chunk de position, ce qui permet de visualiser l'effet des chunks de position graphiquement.

Partie encodage /décodage

La compression se fait véritablement lors de cette étape. Nous avons réalisé deux types de compression mais il est possible d'en réaliser d'autres avec de l'imagination et plus de temps.

Il y a deux phases :

- L'optimisation qui consiste à trouver de nouveaux chunks à créer pour supprimer des informations redondantes dans plusieurs stimuli.

- La factorisation qui consiste à appliquer les chunks aux stimuli. Il arrive avec des chunks non-canoniques (c'est-à-dire des chunks composés de plusieurs autres chunks) que certains chunks soient en concurrence avec d'autres. Il faut alors choisir ceux qu'on applique car l'application de l'un annule celle de l'autre. Cette phase contient aussi ce processus de choix.

L'optimisation et la factorisation sont réalisés à tour de rôle juqu'à ce qu'aucune amélioration ne soit plus possible.

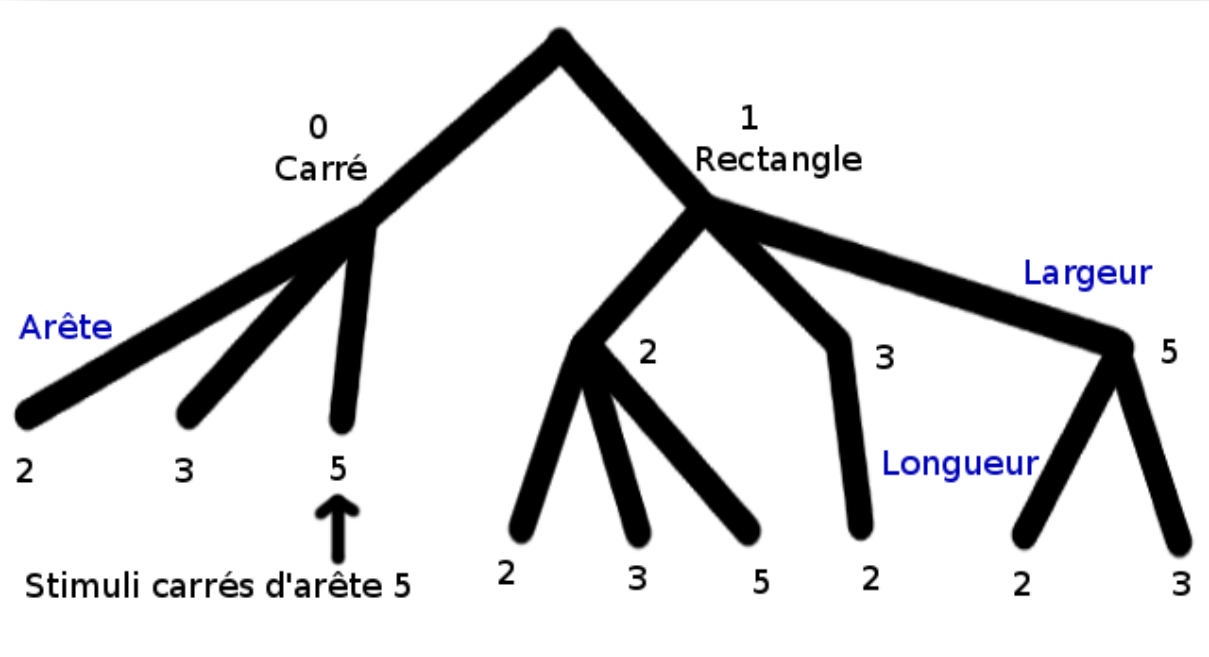
Encodage selon les formes et les dimensions

L'encodage selon les formes et dimensions est le plus simple étant donné qu'il n'y a qu'une possibilité pour le choix des chunks et que les chunks sont canoniques. La phase de factorisation est alors réduite à remplacer les chunks nouvellement crées dans les stimuli/chunks.

La première phase de cet encodage consiste à créer les chunks primitifs et à placer dans les stimuli (qui deviennent alors des stimuli/chunks) les identifiants d'un de ces chunks.

Ensuite vient la phase de regroupement par dimension pour les stimuli de même forme. Dans un souci de garder une faible complexité dans notre algorithme, nous avons opté pour un arbre n-aire permettant de trier facilement les stimuli selon leur forme, puis selon leur dimension. Les stimuli sont d'abord séparés en deux branches : 0 pour les carrés, 1 pour les rectangles (et plus si nous traitions plus de formes), puis les stimuli carrés sont classés selon leur arête, et les stimuli rectangles sont classés dans un premier niveau selon leur largeur, puis selon leur longueur. De cette manière nous pouvons facilement récupérer les stimuli de même forme et même dimension.

Voici à quoi ressemble l'arbre en question :



En itérant sur les feuilles de l'arbre, on peut alors créer des chunks dès qu'une feuille contient au moins deux stimuli. A partir de deux stimuli de même dimension, la création de chunk devient forcément intéressante puisqu'on a alors un gain d'au moins 32-N pour les carrés et 64-N pour les rectangles (et éventuellement une plus grande valeur - N pour d'autres formes plus complexes) avec N valant moins de 32 bits.

Dans notre cas où N = 5 bits sur lesquels sont codés les identifiants de chunks, le gain est d'au moins 32-5 = 27 bits pour les stimuli carrés et au moins 64-5 = 59 bits pour les stimuli rectangulaires, donc nous créons des chunks dès qu'il y a deux stimuli de même forme et même taille.

Le chunk crée contient alors une référence vers un chunk primitif pour la forme et les valeurs de dimension. Les stimuli concernés sont alors débarrassés de leurs valeurs de dimension et font référence au chunk nouvellement crée.

La phase de factorisation consiste ici simplement dans le remplacement des chunks de dimension parmi les stimuli, elle est faite donc en même temps que l'optimisation.

Encodage selon les positions

Cette partie est plus complexe car il est possible de trouver plusieurs possibilités d'assemblage de stimuli pour faire des chunks les regroupant, il faut alors tester toutes les possibilités et garder celles qui font gagner le plus de place. Tester la totalité des

combinaisons possibles entre tous les chunks serait de trop grande complexité et n'est pas envisageable. Il faut trouver des approches plus intelligentes. Dans sa thèse « "Modélisation cognitive computationnelle de l'apprentissage inductif de chunks basée sur la théorie algorithmique de l'information », Vivien Robinet indique qu'il est possible par exemple de limiter la profondeur de tests dans le total des possibilités, ou encore de limiter les tests en limitant le temps de calcul.

Nous avons choisi ici de tester seulement un nombre réduit de chunks rassemblés (seulement deux à deux et non pas n à n) en vérifiant dans la partie factorisation si il est possible de simplifier des chunks de position contenant deux chunks de position en un chunk à quatre positions, et ainsi de suite (cette partie de factorisation n'a pas pu être implémentée par manque de temps).

Nous avons également choisi de réaliser le regroupement de stimuli avec seulement des stimuli de même forme et taille, donc faisant initialement référence au même chunk de dimension. Ceci permet alors de supprimer un des stimuli du couple en n'en gardant qu'un seul pour les coordonnées 2D de son point, et d'avoir l'information de l'autre dans le chunk crée à l'aide d'un vecteur de déplacement (également deux coordonnées x et y) par rapport au premier chunk.

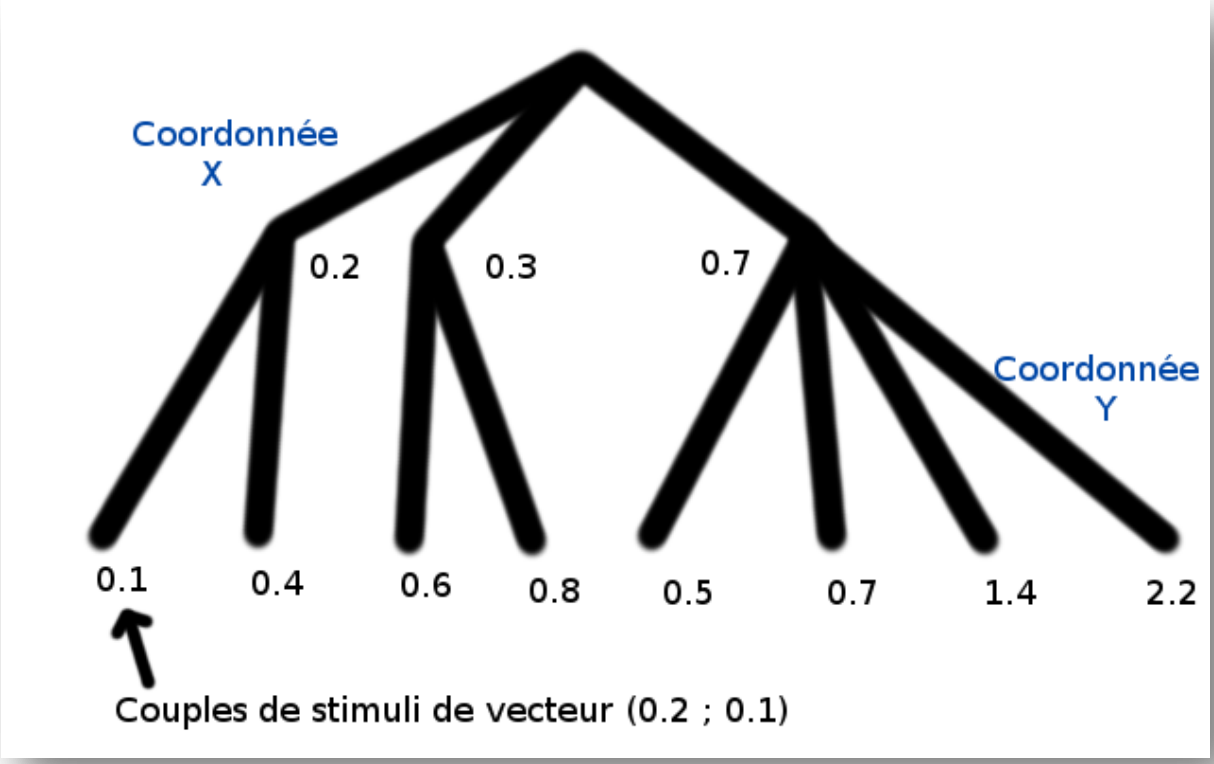
Ce choix est motivé par le fait que la représentation graphique est plus facilement compréhensible en choisissant de rassembler des stimuli de même dimension et forme, et que c'est une technique de compression efficace.

Les composantes des coordonnées étant toujours codées sur 32 bits, on remarque qu'il est inintéressant de créer un chunk de position pour regrouper seulement un couple de stimuli (perte de N+5 bits) mais qu'il devient en revanche intéressant d'en créer un dès qu'il y a au moins deux couples avec le même vecteur de déplacement (gain d'au moins 64-5-N bits, N valant en général 5 et au maximum 32).

Il faut alors, dans la phase d'optimisation, itérer sur les paquets de stimuli de même forme et taille (l'arbre de la précédente compression nous est ici très utile) et pour chacun de ces groupes il faut tester toutes les possibilités d'association deux à deux, en faisant attention à ne pas retester deux mêmes stimuli deux fois (par exemple AB et BA comme couples avec A et B des stimuli).

Chaque possibilité testée est stockée dans un nouvel **arbre des possibilités** n-aire selon les valeurs du vecteur de déplacement, avec la composante x sur le premier niveau de branches et la composante y sur le deuxième niveau, de manière à avoir des feuilles regroupées selon le même vecteur de déplacement.

Voici une représentation graphique de cet arbre des possibilités :



Les feuilles contiennent des objets rassemblant des couples de stimuli et un vecteur de déplacement avec les deux composantes x et y.

Nous choisissons, à chaque itération, la branche de l'arbre de possibilités contenant le plus de couples car il est possible d'avoir des chunks « concurrents », c'est-à-dire dont l'application rendrait inapplicable celle d'autres chunks de même type. Cette technique ne permet malheureusement pas d'avoir le meilleur taux de compression possible. Il aurait fallu à chaque fois remplacer les chunks dans les stimuli et tester le gain, puis revenir en arrière pour les autres tests. C'est la méthode des retours en arrière décrite par Vivien Robinet.

A chaque itération, un chunk au plus est crée par paquet de couples de stimuli de même vecteur de déplacement (donc par branche de l'arbre initial). L'arbre de classement des stimuli selon leur forme et dimension est alors recrée pour prendre en compte le fait que dans certaines branches, des couples ont disparu (l'un des stimuli supprimé totalement, l'autre transformé en stimulus faisant référence à un chunk de position), et le fait que de nouvelles branches font apparition (les stimuli transformés et non supprimés doivent être traités ensemble en essayant de les mettre en couple entre eux).

Lorsqu'il n'y a plus aucun chunk crée, on arête l'itération et la compression est terminée.

Partie stockage binaire

La traduction des données sous forme binaire permet d'optimiser au maximum la taille occupée par ceux-ci sur le support de stockage. Il s'agit de mettre le moins d'informations possibles de manière à pouvoir reconstituer le fichier initial à partir d'un fichier hautement compressé, et en même temps il faut qu'il y ait suffisamment d'informations pour que le fichier compressé puisse rester exploitable y compris si le logiciel évolue. Il faut donc trouver un compromis.

Par exemple nous avions initialement pensé à coder les identifiants des chunks sur 5 bits (ce qui faisait 32 possibilités de chunks différents) en pensant qu'il suffirait d'augmenter ce nombre si jamais il fallait plus de chunks pour un fichier donné. Le problème soulevé est alors que si jamais on créait plusieurs fichiers avec un nombre de chunks différents, certains stockés sur 5 bits, d'autres sur 6 bits (car plus de 32 chunks), alors à la lecture il serait impossible de savoir quel nombre de bits il faut lire pour les identifiants de chunks.

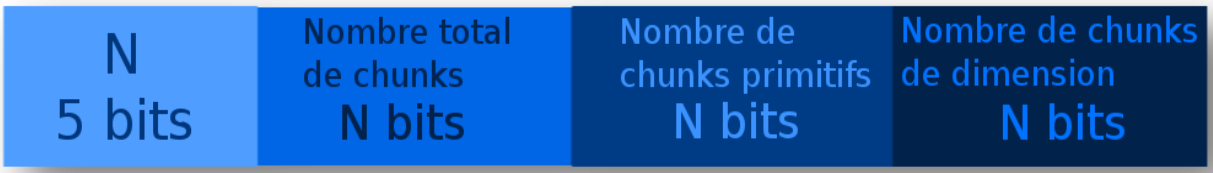
La solution est d'avoir une entête au début de chaque fichier binaire indiquant sur combien de bits sont codés les identifiants de chunks. Cette valeur non-variable est sur 5 bits mais on peut raisonnablement estimer que le nombre de bits sur lesquels sont codés les identifiants de chunks ne dépassera pas 32, autrement dit il n'y aura pas plus de 2^32 chunks différents pour un fichier donné.

Le shéma suivant représente la structure générale que nous utilisons pour le fichier binaire crée :

Entête

L'entête contient une valeur sur 5 bits indiquant la taille de codage des identifiants de chunks, notons cette valeur N. Puis le nombre de chunks primitifs sur N bits, puis le nombre de chunks de dimension sur N bits, et enfin le nombre de chunks totaux sur N bits.

Voici une représentation graphique de la structure binaire de l'entête :



Chunks

Dans le cadre de notre avancée dans le projet et des méthodes de compression implémentées, nous avons trois types de chunks :

Les chunks primitifs
Les chunks de dimension
Les chunks de position

Les chunks primitifs sont des chunks représentant uniquement une forme primitive (par exemple : carré, rectangle) et n'ayant aucune autre caractéristique. Les stimuli lus du fichier texte et stockés en tant qu'objets, sans être davantage compressés, contiennent une référence vers un chunk primitif qui indique leur forme. Les chunks primitifs doivent être connus du programme car il faut qu'il sache comment les traiter et les représenter, donc nous avons décidé de considérer que le programme connait. Il y a une liste en dur représentant les différentes formes primitives possibles avec des identifiants assignés. Les chunks primitifs se trouvent au début du fichier binaire, juste après l'entête. On trouve les identifiants de ces chunks, correspondant à l'identifiant se trouvant en dur dans le programme. De cette manière on peut, avec deux chunks primitifs rectangle et carré, placer le rectangle avant le carré, le rectangle après le carré, placer seulement le carré ou seulement le rectangle.

Voici une représentation graphique de la structure binaire des chunks primitifs :

Les chunks de dimension sont des chunks canoniques comme les chunks primitifs. Ils font référence à un chunk primitif et contiennent en plus une ou plusieurs valeurs permettant de fixer la taille de la forme.

Voici une représentation graphique de la structure binaire d'un chunk de dimension :

Les chunks de position sont des chunks non-canoniques, c'est à dire qu'ils sont composés de plusieurs autres chunks. Ils permettent de regrouper plusieurs chunks selon leur position relative. Nous avons traité ici uniquement les cas de chunks de même forme et taille dont la position relative se répète au sein des stimuli.
A partir du moment où nous avons des chunks non-canoniques, nous pouvons avoir des conflits au sein des chunks. Certains chunks une fois appliqués à une liste de stimuli rendent inopérands l'effet d'autres chunks. Ce conflit est résolu dans la partie factorisation de l'encodage, qui essaye de trouver dans le cas d'un conflit, la meilleure combinaison de chunks à utiliser pour avoir une compression maximale.

Les chunks de position font référence à un chunk de dimension suivi de couples de valeurs représentant des vecteurs de déplacement. Ils ont d'abord une valeur indiquant le nombre de chunks rassemblés (et donc le nombre de couples de valeurs de déplacement à lire), puis l'identifiant du chunk de dimension, et enfin les valeurs.

Voici une représentation graphique de la structure binaire d'un chunk de position :

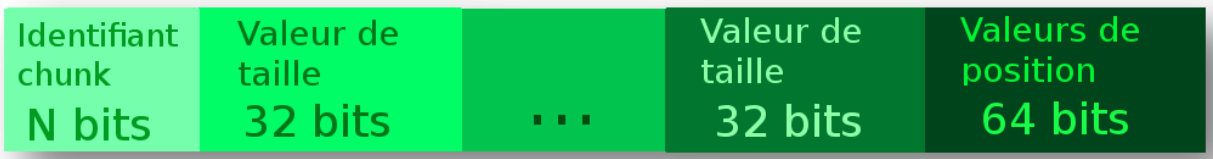
Tout comme nous avons indiqué au début de chaque chunk de position le nombre de chunks qu'il contenait, nous aurions pu également indiquer au début de chaque chunk de dimension le type de forme et donc le nombre de paramètres contenus dans ce chunk. Ceci aurait permis de mélanger les chunks de position et les chunks de dimension dans l'ordre d'apparition , mais la compression s'en trouverait alors affaiblie. Nous avons préféré indiquer dans l'entête le nombre de chunks de dimension, de manière à les traiter séparément des chunks de position lors de la lecture et de l'écriture du fichier binaire.

Stimuli

Les stimuli sont placés à la suite des chunks. Ils représentent les données alors que les chunks représentent le dictionnaire.

Les stimuli sont composés d'un identifiant vers un chunk, codé sur N bits, suivi d'éventuelles valeurs de taille sur 32 bits chacune (dans le cas où il n'y avait pas de stimulus de même forme et même taille et où un chunk pour regrouper leur taille n'a pas été crée), et de deux valeurs x et y sur 32 bits chacune (64 bits pour la paire de valeurs) représentant la position d'un point pour repérer ce stimulus sur le plan 2D. Dans le cas où ce stimulus serait regroupé avec un autre stimulus à l'aide d'un chunk non canonique, ce stimulus serait détruit et son information serait alors contenue dans un seul stimulus faisant partie du groupe réuni et du chunk de position.

Voici une représentation graphique de la structure binaire d'un stimulus :



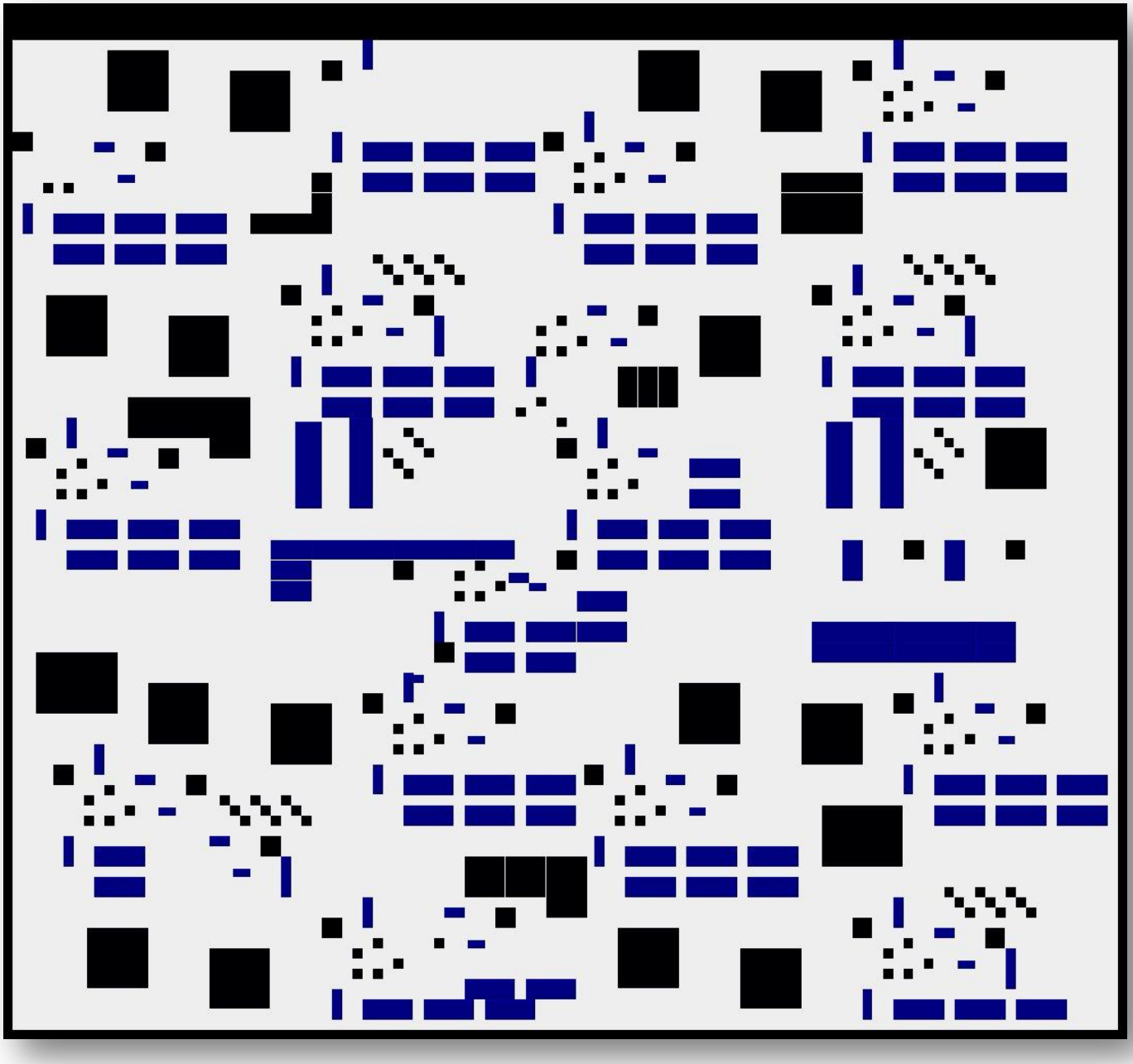
Conclusion

Ce projet nous a permis d'entrevoir certaines techniques de compression de fichiers. On peut déduire de ce que nous avons réalisé que la meilleure forme de compression pour une donnée est celle qui a spécialement été conçue pour celle-ci, en prenant en compte toutes ses caractéristiques et comptant les données nécessaires bit par bit.

De même nous avons appris qu'en ayant plus de temps et plus de puissance de calcul et place en mémoire, il était possible de faire des tests plus exhaustifs et donc de mieux compresser, alors qu'avec moins de temps et de puissance/mémoire, il était possible de compresser tout de même mais sans tester toutes les possibilités et donc avec un taux de compression moindre.

Nous avons également appris qu'il faut parfois faire des compromis entre le gain de compression et l'assurance d'avoir un fichier compressé suffisamment « interopérable » avec des versions futures du logiciel. Par exemple en plaçant des informations dans l'entête.

Annexe 1 : exemple de stimuli non compressés



Annexe 2 : exemple de stimuli compressés par leur dimension